THE IMPLEMENTATION OF APL\360

by

L. M. Breed
R. H. Lathwell

IBM Watson Research Center
Yorktown Heights, New York

Presented at the ACM Symposium
on Experimental Systems for
Interactive Applied Mathematics.

THE IMPLEMENTATION OF APL\360

Introduction

APL\360 is an experimental, conversational System/360 implementation of APL, the Iverson language*. It provides fast response and efficient execution to a large number of typewriter terminals. With 40 to 50 terminals connected and in normal use, each with a block of storage (called a workspace) allocated, reaction time (defined as the time from completion of an input message until the user's program begins execution) is typically 0.2 to 0.5 seconds. At the terminal this is manifested by nearly instantaneous response to a trivial request. Under these conditions, the CPU is executing user programs about 75% of the time and supervisor overhead and I/O wait time amount to less than 1%. The APL processor is interpretive; however, because of the efficiencies afforded by array operations, program execution is often one-tenth to one-fifth as fast as compiled code.

---

*APL\360 as it appears to the user is described in (1). As of November 1967, APL\360 is an IBM Research proprietary program, and is not available for distribution.

APL\360 is currently running on a System/360 Model 50 with 262,144 bytes of core storage, a 2314 Direct Access Storage Facility, and two 2702 Transmission Controls to which IBM 1050 and 2741 Communication Terminals are connected via telephone lines.

System Characteristics

The APL\360 program is divided into two parts, the supervisor and the APL interpreter. The supervisor is responsible for disk and typewriter I/O, scheduling the CPU and user storage (which is normally on disk and is swapped into main storage as required), maintenance of the interval timer, and processing APL system commands. The interpreter is a read-only program which executes APL statements and relays system commands to the supervisor.

APL\360's performance can be traced to the following salient features, which arose naturally from the characteristics of APL and the implementation's initial design as a dedicated system.

1. The supervisor has complete control over system resources, so that all forms of storage can be allocated according to a single overall

strategy, and swapping can take place at full disk transfer rate.

2. The system design was directed by an advance analysis of the nature of APL user programs. For instance, execution speed of array operations is enhanced at the expense of speed in APL program loops. Also, processing of an input line may require access to many parts of a workspace, so the supervisor provides for efficient monolithic transfer of workspaces between disk and core.
3. System overhead, swap time, and address relocation problems are minimized, and supervisor-interpreter communications simplified, by the adoption of specialized programming conventions in the interpreter.

Workspace Organization

The APL workspace, illustrated in figure 1, is designed to minimize the amount of information which must be transferred during a swap operation. 1000 bytes of storage at the lower end of the workspace and 3000 bytes at the upper end are fixed, and the rest is allocated dynamically.

To keep unused storage contiguous, allocation occurs from both ends of the workspace toward the center. The execution stack, created by the system, extends downward from the bottom of the symbol table. The variable SVI points to the 'top' of this inverted stack. Storage for data and programs created by the user are allocated in blocks of arbitrary size called M-entries, which extend upward from the low fixed area. The variable MX indicates the end of this storage. During a swap, only the portions of the workspace below MX and above SVI need be transferred. This typically amounts to about half the workspace size.

An M-entry may hold an entire array, a statement in a user program, or other data of arbitrary size and structure. It requires only the space needed for the data, plus two words of storage allocation overhead. One word holds the length of the M-entry; the other is a pointer to an entry in the symbol table or execution stack, which in turn points back to the M-entry. References to a variable or user program are always to the symbol table entry, which has a fixed location, rather than to the M-entry itself. Thus, the symbol table or stack entry is the only

pointer that must be modified when an M-entry is relocated, for instance by garbage collection.

The Interpreter

An APL statement transmitted from a typewriter is immediately converted symbol-for-symbol to an internal form called a codestring, in which variables and function identifiers are represented by 16-bit symbol table pointers, special characters by 8-bit internal codes, and constants by a code, a count, and their converted hexadecimal values. No other preprocessing, such as conversion to Polish, is done. APL is sufficiently close to Polish prefix form, because of its right-to-left operator precedence, that further conversion would yield little improvement in processing speed. It would, moreover, complicate the conversion back to input form that is done for typewriter display.

Figure 2 shows the codestring corresponding to the APL statement

```
A ← EVTP[2 3 7]*0.5
```

where *A* is global, occupying a symbol location 176 bytes from the end of the symbol table, and *EVTP* is the fourth local variable. Note that `2 3 7` is carried as a single, vector, constant.

Long and short syllables in a codestring (except for constants) are differentiated by the rightmost bit of each syllable, because interpretation proceeds from right to left. The right-to-left interpretation was suggested by APL's operator precedence, but interpretation proceeding left to right might have been as appropriate.

Because APL block structure is static, local variables may be bound at input, and no symbol table searches are required during execution. In a codestring, a global variable is represented by a negative number -- the address of its symbol table entry relative to the end of the symbol table. A local variable or parameter to a function *F* is represented by a small positive number interpreted as an address in the execution stack relative to the position of function-call information for *F*. The stack entry thus addressed serves during the life of the function call as a symbol table entry, pointing to the M-entry which is the value of the local variable.

An earlier implementation kept constants in a special symbol table and represented them in codestrings by symbol table pointers. It was soon discovered that without an elaborate garbage collector, the

workspace quickly became filled with constants no longer needed. In the present implementation, the constants automatically vanish when the codestring carrying them is deleted.

Syntax analysis is performed using Conway transition diagrams[2]. Although APL does not require such a general method, transition diagrams have made experiments with the syntax of the language particularly easy. Figure 3 shows the diagram for the syntax of a 'basic'. Passage through the diagram via paths corresponding to successive codestring elements indicates a successful syntax analysis of a 'basic'. 'List' is the name of a syntactic quantity defined by another diagram which may use the 'basic' diagram recursively. The boxed letters indicate interpretation rules, or actions to be taken when the corresponding paths are successfully traversed.

The uniformity of APL operator definitions lets a single control program perform nearly all operator execution. Conversions between the three numeric data types (logical, integer and long floating point) are done automatically by the operator execution control program, and with a few exceptions it is impossible for a user to determine the representation of his data.

Execution is strongly biased toward operations on arrays. The overhead time for the syntax analysis and setup for evaluation of a simple expression is two to three milliseconds, while execution time for each scalar element is typically 40 to 250 micro-seconds.

Main Storage Management and Disk Swapping

Core storage for users is divided into fixed-size areas (in the present system, 36000 bytes), each of which may hold a workspace. A minimum of two such areas is required, but system performance, in terms of response time and reduced I/O waits, improves significantly with three or more areas. The first two areas are necessary to permit I/O buffering during swap operations; any additional areas increase the probability that some workspaces in core are active (ready to run) and therefore that the CPU is not idle.

With minor exceptions, all pointers to storage locations within a workspace are kept relative to the beginning of the workspace. Because of this, and the fact that their size is fixed, workspaces may be transferred between any disk and any main storage area with no relocation or storage fragmentation

problems. This makes possible a very efficient and simple time-sharing scheduler. Absolute addresses may, of course, be produced during program execution. By convention, the interpreter acts on a supervisor request for end-of-quantum only when program execution requires no absolute addresses (typically every 0.1 to 10 milliseconds) and may therefore be suspended.

The scheduler attempts to keep as many active workspaces as possible in main storage, while minimizing unnecessary swaps. No swapping takes place unless some workspace on the disk is active. If such a workspace exists and all core areas are occupied, some workspace in core must first be written onto the disk. An inactive workspace (one that is awaiting completion of a typewriter message) in core is selected to be written; or, if all workspaces are active, the one which has been given the most CPU time since it was last brought in is selected. A workspace is always written to the area from which a workspace was last read, so disk arm motion is required only in read operations.

An undesirable situation arises when more workspaces than can fit into main storage are engaged in long computations. Under this condition, the swap

algorithm causes continuous swapping of active workspaces, which reduces system performance due to storage interference and increases response time. A modification to the algorithm, not yet implemented, would arbitrarily reduce the swap rate to one swap every ten quanta unless an input message had just been completed for some workspace not in main storage. CPU service is thus concentrated on a subset of the active workspaces. The subset receiving the concentrated service changes slowly as swapping occurs.

Error Recovery

The treatment of user-program errors has been described in (1). Recovery from such errors involves immediate discontinuation of execution, cutting back the execution stack to the level of the most recently called function, and requesting typewriter input. Error recovery is rarely unsuccessful, because the integrity of the symbol table, stack, and storage allocation information is protected during execution by array-bounds checking of all subscripted variables.

Recovery from system errors is accomplished differently. To prevent an interpreter bug from taking down the system, all main storage except for the workspace being executed is store-protected, and infinite loops

are forcibly terminated by the interval timer routine. Any program check within the interpreter causes the error routine to print the general registers on the user's typewriter (which will show the particular sequence of inputs and outputs leading to the difficulty) and attempt an error recovery as above. Other users remain unaware of the program check; in fact, it is impossible for a user to take down the system without assistance from the machine operator.

A noteworthy distinction of a conversational as opposed to a batch-processing system is that the environment of a user program (in APL\360, the workspace) is maintained for long periods of time -- months instead of seconds -- and that usually this environment contains the only record of the user's work. In a batch-processing system, if a wild store modifies a program, only one run is lost. One simply corrects and resubmits the deck. In a conversational system, however, it is critically important to guard against system or user errors that may compromise the integrity of the environment and remain undetected for several days.

Another consequence of this distinction is that program and data representations not usually a concern of the user (internal table formats or collating sequences, for example) are very difficult to change because they are carried semipermanently as part of the user program environment. If they are to give continuity to the environment, different versions of the system must, in a sense, be compatible at the bit level rather than the source language level. Generally speaking, the more pre-execution processing of programs is done, the more difficult it becomes to introduce system changes.

System Measurement

While running, APL\360 gathers statistics on its performance and on user behavior. These statistics are made available (via a special operator) as data to APL programs, permitting comprehensive and easily modified monitoring of the system. Figures 4 and 5 are examples of such measurements. Figure 4 shows the distribution of reaction times for an average load of 40 terminals over a period of 36 hours. Figure 5 shows the distribution of keying time, defined as the interval between requesting and receiving an input message, for the same period.

Acknowledgements

We wish to acknowledge our very fruitful collaboration with Philip S. Abrams, Stanford University, on an earlier implementation. We are also grateful to Luther Woodrum, IBM, for his continuing contributions to the implementation. Lastly, APL\360 owes much of its superior time-sharing performance to Roger D. Moore, of I.P. Sharp Associates, Toronto, who was principally responsible for the supervisor. Its design has not been described to the extent it deserves.

References

[1] Falkoff, A. D., and Iverson, K. E., "The APL\360 Terminal System". Proceedings of the ACM Symposium on Interactive Systems for Experimental Applied Mathematics.

[2] Conway, M. E., "Design of a Separable Transition-Diagram Compiler", C. ACM 6, 396 (1963).

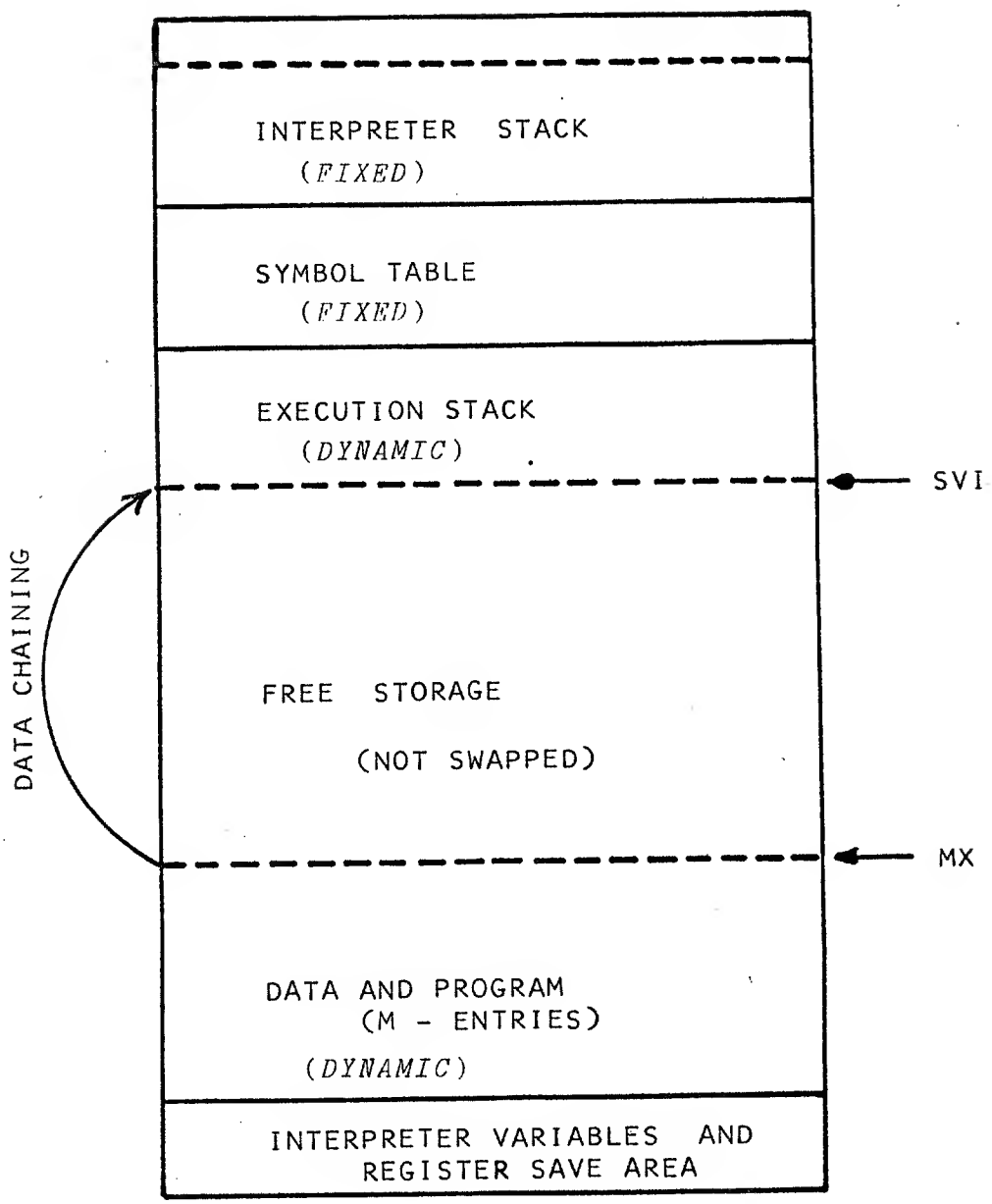

FIGURE 1: APL\360 WORKSPACE ORGANIZATION

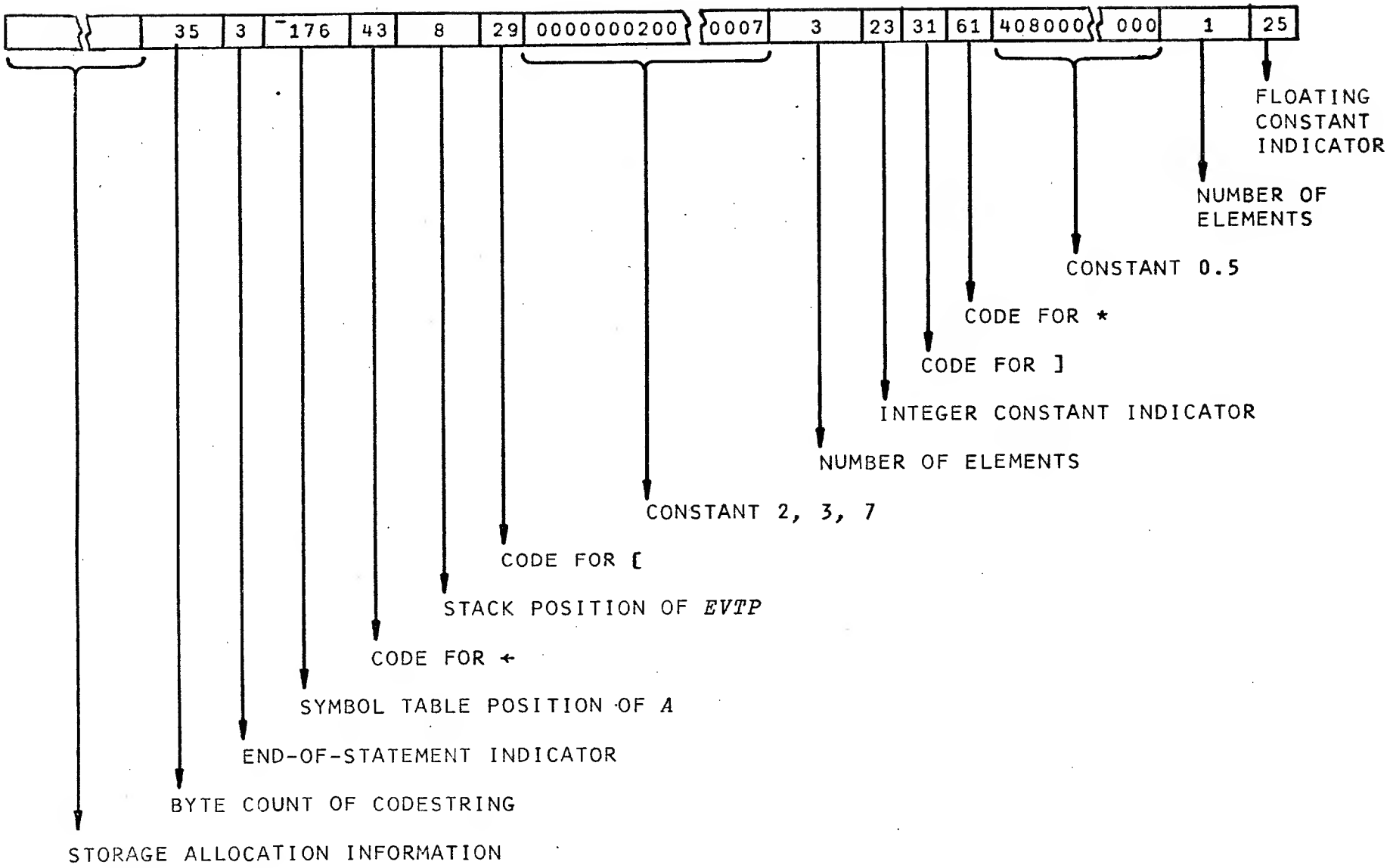

FIGURE 2: CODESTRING FOR THE STATEMENT $A \leftarrow EVTP[2\ 3\ 7]*0.5$

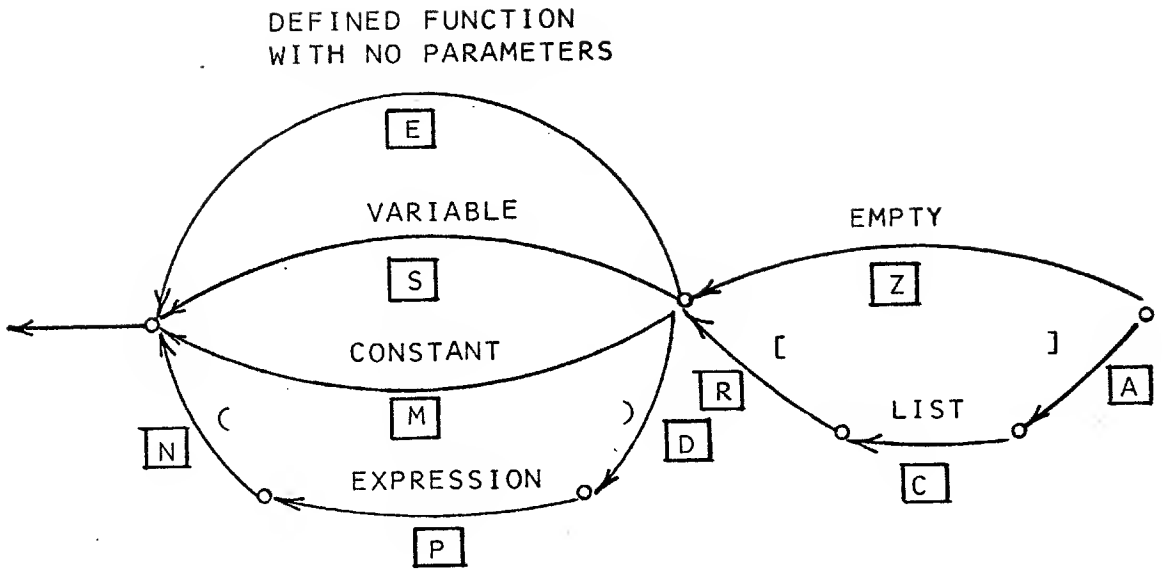

FIGURE 3: TRANSITION DIAGRAM FOR THE SYNTACTIC QUANTITY 'BASIC'

```
50 60 PLOT CRESPONSE VS (⍳50)÷10
```

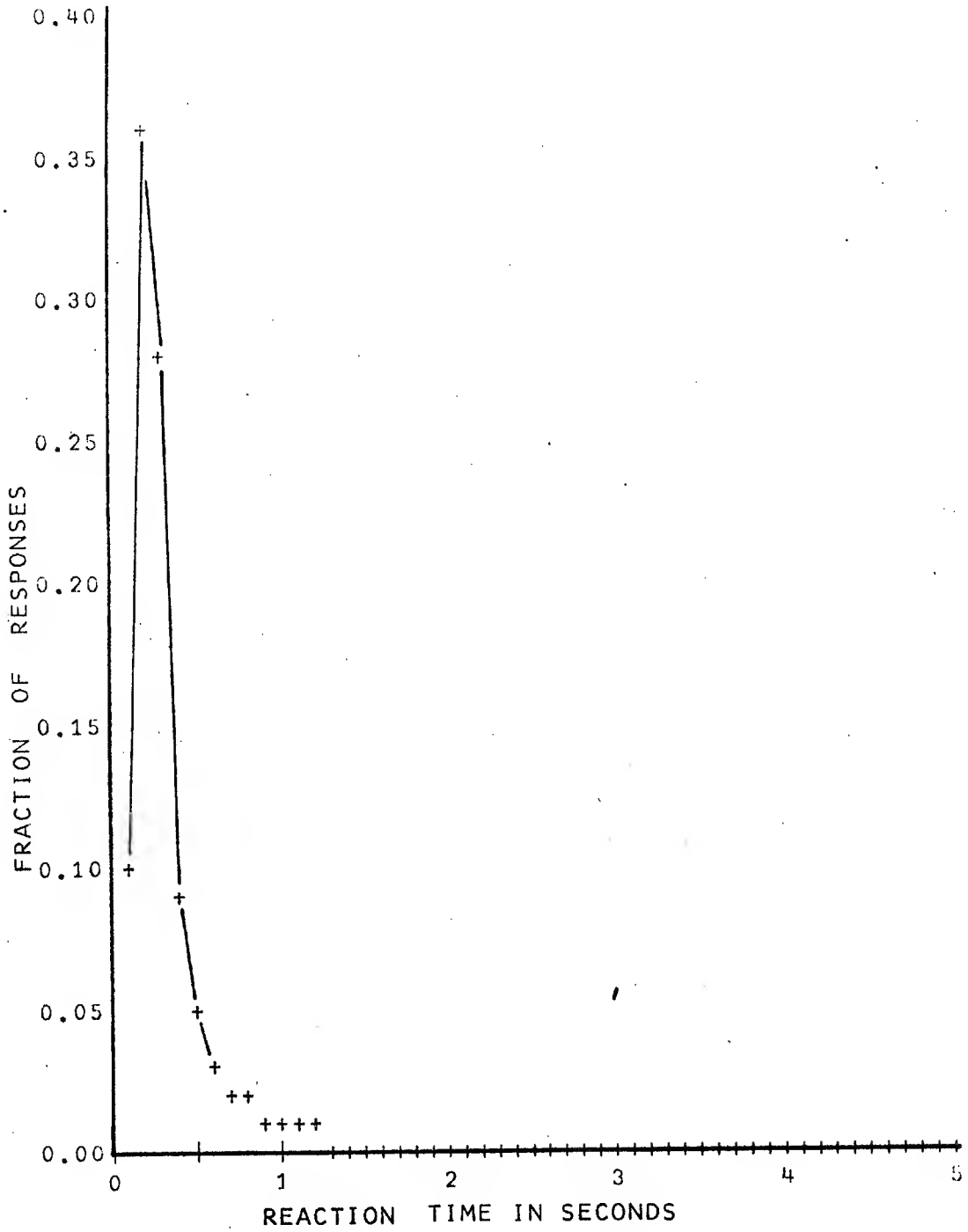

FIGURE 4: APL\360 REACTION TIMES FOR 40 USERS
(SAMPLE : 146,341)

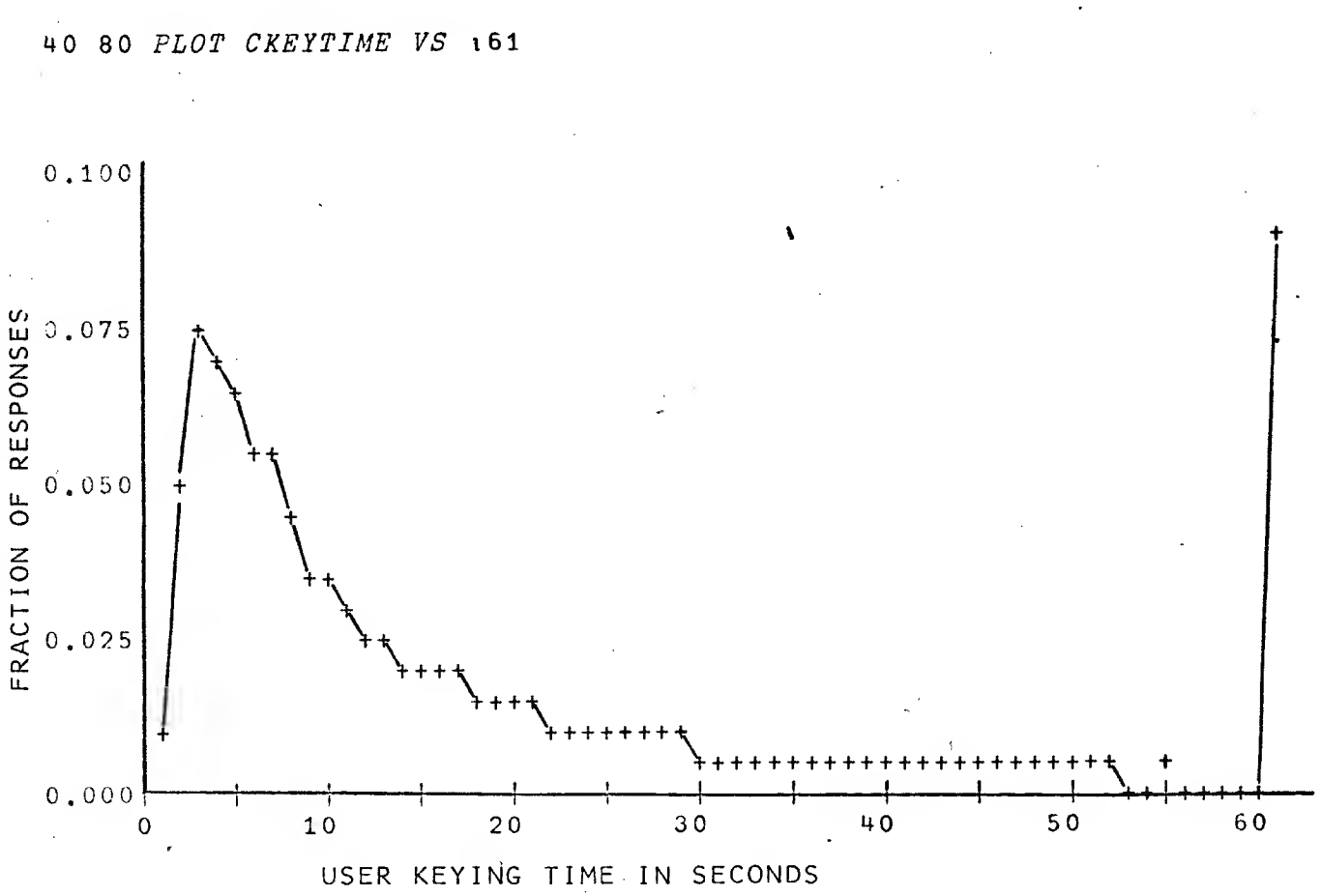

FIGURE 5: APL\360 USER KEYING TIME

(SAMPLE : 146,341)